Evaluating the Use of Model-Based Requirement Verification Method: An Empirical Study

Munmun Gupta, Daniel Aceituna, Gursimran S. Walia, Hyunsook Do
North Dakota State University, Fargo, ND, USA
{munmun.gupta, daniel.aceituna, gursimran.walia, hyunsook.do}@ndsu.edu

***Abstract*—Requirements engineering is a critical phase in software development that describes the customer needs and the specifications for the software solution. Requirements are gathered through various sources and the output is a list of requirements for a software product to be developed, written in Natural Language (NL). NL requirements are fault prone because stakeholders can interpret NL differently due to the inherent imprecision, ambiguity, and vagueness of NL. To address these problems, a model-based requirements verification method called NL to state transition diagram (STD) is proposed. This paper evaluates the ability of the NLtoSTD method to detect faults when used on NL requirements and to improve the software reliability. Overall, the result shows that the NLtoSTD is an effective requirements verification method.**

I. Introduction

To ensure software reliability, it is important to detect and prevent different types of faults during the development of various software artifacts. Requirements are gathered from different stakeholders (technical and non-technical) and recorded in natural language (NL), that describes the customer needs and the specifications for the software solution. The output of this phase, a *software requirement specification* (SRS - a means of communication among stakeholders), is especially fault-prone due to the inherent imprecision, ambiguity, and vagueness of NL. Requirement faults if undetected propagate to the later phases where they are difficult to find and fix [1-3].

To ensure high-quality requirements, numerous fault-based verification approaches have been developed and validated for fault-detection effectiveness [3, 8-11]. In particular, *software inspection*, have been empirically validated [3, 9] for early detection of faults in software artifacts. However, it is estimated that the software development effort is still spent on fixing problems that should have fixed early in the lifecycle [1, 2]. This rework stems from the fact that inspectors can have different interpretations of the requirements and may not notice the ambiguities and inconsistencies among other problems.

Model based approach, if applied to NL requirements can be used for verification of NL specifications [6, 10]. However, building a model from NL requirements is highly subjective. Consequently, an erroneous translation of NL requirements can result in the wrong model due to the inherent incompleteness and ambiguities of NL [11] and, thus, can eventually produce software that stakeholders do not want. To address this, several researchers have proposed modeling techniques using an automated NL translation approach [4,6,10,12]. These methods include approaches based on translating goals to state machines [4], and scenarios to state machines [12]. Automation can certainly improve the translation process, but complete and error-free automation of this process is not possible because, often, NL requirements can be interpreted in multiple ways..

To address this problem, we propose a method that translates NL requirements into a State Transition Diagram (STD) in an incremental manner (NLtoSTD) and expose ambiguities, incompleteness, and inconsistencies in NL requirements. The NLtoSTD is carried out in two steps, where the first step turns each NL requirement into a STD building block (*NLtoSTD-BB*) and the second step then construct the STD using the STD-BBs (*STD-BBtoSTD*). Requirements engineers and stakeholders can detect faults during each step (NLtoSTD-BB and STD-BBtoSTD) and direct mapping from NL to model is preserved in the translation process. Each NL requirement becomes a segment of the STD so that adjustments made to the model can be directly made to the requirements, and visa-versa. The results from the previous study [5] validated the NLtoSTD-BB method and helped us make revisions. This paper presents an empirical study that evaluates the fault-detection ability of the revised NLtoSTD-BB method, and extends the research by evaluating the fault-detection ability of the STD-BBtoSTD method (used for the first time). Therefore, the complete NLtoSTD method (i.e., *NLtoSTD-BB* + *STD-BBtoSTD*) is evaluated in this paper.

II. Background

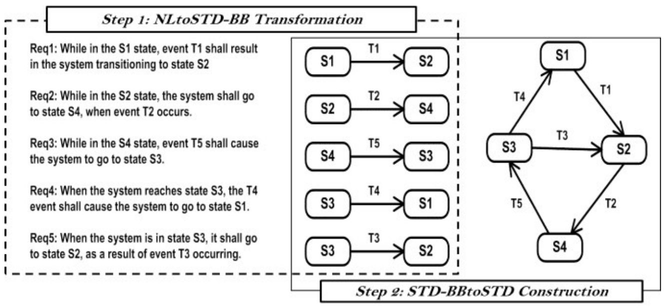
This section describes the basic concepts of the NLtoSTD method, the revised NLtoSTD-BB, and STD-BBtoSTD step.

A. NLtoSTD: Basic Concepts

The basic idea of our NLtoSTD method is to translate the NL requirements into an STD, so that the ambiguity, incompleteness, and inconsistencies or any problem) in the NL requirements can be easily detected. The NL to STD translation first translates NL requirements into STD-BBs (*NLtoSTD-BB*) and then creates an STD using the STD-BBs (*STD-BBtoSTD*). A high-level overview of the NLtoSTD method is shown in Fig. 1. Fig.1 highlights the "*NLtoSTD-BB*" translation (Step1) and the "*STD-BBtoSTD*" construction (Step2). We hypothesize that the NLtoSTD helps discover the problems in the NL requirements by examining the individual STD-BBs and the resulting STD.

As shown in middle part of Fig. 1, the three elements that make up a STD-BB (i.e., *current state* (*Sc*), *next state* (*Sn*), and *transition* (*T*)) are precisely extracted from an individual requirement. The selection of these elements was based upon the characteristics of an ideal requirement and an inspection scheme that can help detect the problems that are otherwise left undetected using traditional inspection methods. As illustrated

This work was supported in part by NSF CAREER Award CCF-1149389 to North Dakota State University.


Req1: While in the S1 state, event T1 shall result in the system transitioning to state S2

Req2: While in the S2 state, the system shall go to state S4, when event T2 occurs.

Req3: While in the S4 state, event T5 shall cause the system to go to state S3.

Req4: When the system reaches state S3, the T4 event shall cause the system to go to state S1.

Req5: When the system is in state S3, it shall go to state S2, as a result of event T3 occurring.

Step 1: NLtoSTD-BB Transformation
Step 2: STD-BBtoSTD Construction
S1
S2
S3
S4
T1
T2
T3
T4
T5

Fig. 1. NLtoSTD Method Overview

in Fig. 1, each requirement is stated so that it directly maps to an STD-BB. Each requirement explicitly states its precondition in the form of the current state (*Sc*) and its post condition in the form of the next state (*Sn*). However, typical NL requirements do not explicitly state current and next states, thus a requirement's preconditions and post conditions are often inferred causing ambiguities and incompleteness. Similarly, the absence of the explicit transition (*T*) can cause difference in the interpretations of a requirement by different stakeholders. The NLtoSTD method requires the stakeholders to identify the aforementioned three elements so that they can detect the requirement faults while building the STD and ensures that the requirements are as consistent and concise as possible.

B. Step 1 - NLtoSTD-BB: Application for Fault Detection

The first step of the NLtoSTD method transforms each NL requirement into an individual STD-BB. The STD-BBs act as a formalized version of the NL requirements and can lead to the detection of faults for two reasons: (1) a formalized version of the NL requirements has only one specific interpretation, exposing *ambiguities* in the NL requirements, and, (2) a formalized version exposes *missing* requirements more readily, as compared to a fault-checklist inspection of requirements.

1) Original NLtoSTD-BB [5]

As shown in Fig. 1, each NL requirement is translated into an STD-BB by extracting three elements {Sc, T, Sn}. The basis for this transformation is that a functional requirement should describe an entity transitioning from one state to another. For example, the requirement "*While the car is moving forward, the driver shall be able to stop it by applying the brake*." would describe the Car (an entity) transitioning (T) from moving (Sc) to stopping (Sn), using an STD-BB.

In the above example requirement, the three elements are explicitly stated, yielding definable values for *Sc*, *T*, and *Sn.* In practice, however, requirements often ambiguously imply one or more values for *Sc*, *T*, and *Sn*, thus identifying a value for each element would not be obvious. For instance, the prior requirement may have stated: "*The driver shall stop the car by applying the brakes*." Note that *Sc* is not explicitly stated as "*moving*" but, rather, implied. In our original STD-BB, we used questions marks (*???*) to denote an element that is not documented. Thus, in this example, we would define the three elements as {*Sc: ???, T: Applying Brake, Sn: Stop*}. It may be safe to assume that the car is moving prior to stopping, but it requires an assumption. Undocumented assumptions can be erroneous and can cause serious defects (especially when the developers lack appropriate knowledge of the application domain). In this example, it is not clear whether we assume "*moving forward*," "*moving backward*," or both. It is important to document what may seem obvious, instead of allowing the possibility of an erroneous assumption. Therefore, the NLtoSTD-BB helps to expose undocumented assumptions.

We developed a set of three questions to help users systematically identify the three elements for each requirement during this step [5]. Asking these three questions identifies explicit or undocumented values for {*Sc, T, Sn*}, resulting in an STD-BB. While the ambiguities and incompleteness may not be obvious in the NL requirements, they are made obvious in an STD-BB that stakeholders can work towards its completion.

2) Revised NLtoSTD-BB

The NLtoSTD-BB used during the Sudy 1 showed that the method was significantly more effective than the fault-checklist based inspection, when the subjects correctly extracted the STD-BBs. The variations in performance during study [5] prompted us to re-evaluate the way that the three elements (Sc, T, Sn) were determined. This section dicuss the changes and the revised NLtoSTD-BB method.

Fig. 2 illustrates the revised NLtoSTD-BB method using an example requirement. In the revised NLtoSTD-BB method, the three changes are briefly discussed along with their reasoning.

The ***first change*** is that we explicitly added an entity to a state to represent Sc and Sn as follows: entity (state). Allowing for multiple entities would alleviate the problem encountered with requirements that are not atomic. Separating the concepts of an entity and its given states, also makes it easier to derive Sc and Sn, since the user could first decide which entity is being affected and then determine the entity states before and after the effect. Fig. 2 shows that the revisded NLtoSTD-BB method, can help identify three entities: unit, battery, and user. The ***second change*** in the revised NLtoSTD-BB method is allowing users to make an assumption. As shown in Fig. 2, "*unit (normalOp?)*" has a question mark. This indicates that the user can assume that it is the intended state and label it for follow up. This was done to improve the method's ability to expose ambiguities. The ***third change*** is to allow users to add conditions when they describe the transition (T). This alleviates the problem when a requirement seems to state more than one transition. The revised NLtoSTD-BB method with five elements (*entity*, *entity's current state*, *entity's next state*, *transition*, *condition for transition*) is evaluated in this paper.

C. Step 2 - STD-BBtoSTD: Application and Tool Support:

The final step in the NLtoSTD method is the construction of STD based on the STD-BBs. The idea is to be able to both simulate the behaviours described in the requirements and to

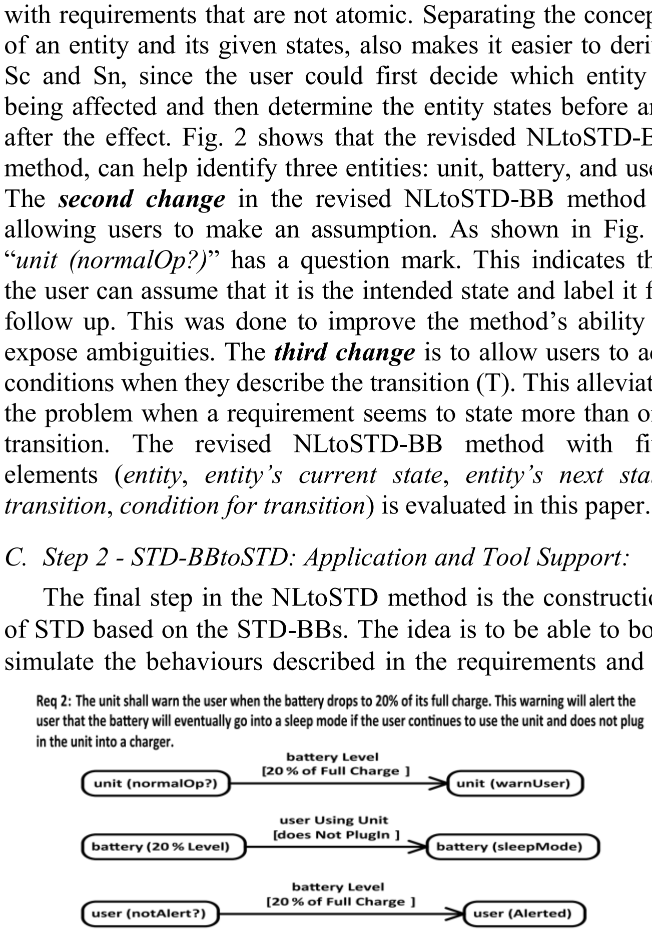

Req 2: The unit shall warn the user when the battery drops to 20% of its full charge. This warning will alert the user that the battery will eventually go into a sleep mode if the user continues to use the unit and does not plug in the unit into a charger.

Fig.2. Revised NLtoSTD-BB

analyze these behaviours through the production of path traversals through the STD. This would potentially expose inconsistent and/or incorrect behaviours, that can be readily traced back to the requirements. An incomplete STD would expose imcompleteness in the requirements verbage. Thus, the STD-BBtoSTD phase exposes potential faults by analyizing the STD's static and dynamic properties.

The STD's behavior was simulated by computer in order to expose faults that may not be evident unless the STD is enacted. There STD can be analyzed automatically by examining the path traversals, for desired behavior. The STD's construction was implemented through a software tool. The user enters the STD-BB data in an Excel spreadsheet, which is then read directly into the tool (by the tool's use of COM automation). The STD is then displayed in a separate window, and can be kept opened as more STD-BB data is being entered. The tool updates the STD, as changes are made to the STD-BB data. This allows the user to view the STD, make changes to the STD-BBs that would correct any STD structural faults, and see the results of those changes in real time. The subject can record faults during this step.

III. Empirical Study

This study evaluated NLtoSTD-BB translation step at finding incompleteness and ambiguities during an inspection of NL requirements. This study also evaluated if additional new faults can be found during the construction of the STD. The complete NLtoSTD method was evaluated using a *repeated-measure design* in which different student teams (with varying number of members) developed requirement documents for different systems. Next, each participant individually inspects the requirement document (that was developed by them) using the NLtoSTD-BB step and kept a log of ambiguous, and missing requirements for respective documents. During the next step, the student individually worked to create STD from the STD-BBs (using an automated tool) and then analyzed the resulting STD to log new faults that were not found previously.

A. *Research Questions and Hypotheses*

The following research questions were investigated:
RQ1: Is the NLtoSTD-BB effective at detecting incomplete and ambiguous requirements during requirements inspection?
RQ2: Does creating the STD model result in detection of the faults in addition to those found during the NLtoSTD-BB?
RQ3: What are the problems faced by the subjects when using the complete NLtoSTD method?

B. *Variables and Measures*

Each subject performed an individual inspection of their requirements document using the NLtoSTD method. Our dependent variables include: *Effectiveness* - # of faults found and *Efficiency* - # of faults found per hour.

C. *Participating Subjects and Requirement Artifacts*

Sixteen computer science graduate students at North Dakota State University (NDSU) worked in teams to develop requirements document for different projects. Some students dropped the course resulting in this irregular size of student teams. There were two phases to this study. ***First***, each team developed a requirement document for a particular software system (Table I). ***Second***, each subject inspected the requirement document developed by their team using the NLtoSTD-BB followed by STD-BBtoSTD method.

TABLE I. ARTIFACTS DEVELOPED BY STUDENT TEAMS

Doc	*Team #*	*No. of Subjects*	*System Description*	*Size (pages)*
A	1	4	Parking lot availability system	25
B	2	4	Web portal for student residence	22
C	3	3	Virtual story board system	28
D	4	2	Matbus application for android	25
E	5	3	Professional development	32

D. *Study Procedure*

The study details are provided in the following subsections.

1) Phase I – Development of SRS documents: The participants divided into 5 different teams of three or four participants developed the requirements documents for their identified software system. Details are provided in Table I.

2) Phase II– Inspection using NLtoSTD: During this step, the students in each team individually inspected their own SRS document using the NLtoSTD method.

a) Training 1 -- NLtoSTD-BB: The participants were first trained on how to map the NL requirements to STD-BBs. Next, the participants were instructed how to document the building block elements on a spreadsheet using few examples. Next, the participants were instructed how to record "*ambiguities*" and "*incompleteness*" or any other requirement faults that are found during the application of NLtoSTD-BB. The subjects were asked to translate few requirements into STD-BBs and document the faults using the same spreadsheet.

b) Step 1-- Inspection using NLtoSTD-BB: The participants used the information from Training 1 and individually inspected their own requirement document using the NLtoSTD-BB translation. This step resulted in a list of 16 individual speadhseets that contained the STD-BB elements and the faults found (one per participant).

c) Training 2 - tool support for STD Creation: During this session, the participants learned about the STD tool. The subjects were instructed how to load the BBs (from step 1) into the tool and then, how to construct an STD from the BBs. The subjects were then instructed to examine the constructed STD. Finally, the participants learned to record the fault type in the fault spreadsheet. To ensure that subjects understood, the subjects practiced these steps through an example system.

d) Step 2 – STD-BBtoSTD and inspectng STD: The subjects constructed the STD diagram from STD-BB. The output of this step was 16 individual STD diagrams (one per participant). The resulting STD diagrams were analyzed for potential incompleteness, inconsistencies in the behaviours, or any other requirement faults. The participants analyzed and recorded the faults during and after the creation of STD. The students also documented the reason and classification of the fault (*incompleteness*, *ambiguous*, *inconsistency* or *other*) in the fault spreadsheet. This step resulted in 16 individual fault lists. Finally, subjects provided feedback about the NLtoSTD-BB and the STD-BBtoSTD. An in-class discussion with subjects helped researchers better understand the results.

E. *Data Collection*

The *quantitative* data included the *ambiguous*, *missing*, and *inconsistent* faults found by each subject in their SRS document during: a) translation of NL requirements to the STD-BBs, and b) construction of STD using the BBs. Each subject was provided 50 minutes during NLtoSTD-BB step and 30 minutes during STD-BBtoSTD step. The timing data was used for analyzing the efficiency values. The *qualitative* data included student's rating of NLtoSTD by answering a multi-question questionnaire based on a 5 point likert-scale. We also collected feedback post-study with participating subjects.

IV. Data Analysis And Results

This section analyzed the data collected during NLtoSTD-BB, STD-BBtoSTD, post-study questionnaire and interviews.

A. *RQ 1: Effectiveness and Efficiency of NLtoSTD-BB*

This section reports the *effectiveness* and *efficiency* of the NLtoSTD-BB during the requirements inspection. Before analyzing the fault data, the researchers determined the validity of the faults for each subject by reading through the fault spreadsheet reported by each participant to remove any false-positives (or if any faults were unclear). Next, the number of "*Missing Functionality* (MF)" and "*Ambiguous Information* (AI)" faults reported by each subject during the application of NLtoSTD-BB for their respective documents were counted.

Since each subject individually inspected their own document, the document (for each team) was inspected by all the subjects belonging to that team. Fig. 4 organizes results by the total number of AI and MF faults found by the member belonging to each team. Main observations from Fig. 4 follow:

- Fifteen out of sixteen subjects found faults (AI or MF) during the NLtoSTD-BB based inspection of their requirements document. The subjects (numbered 6) reported a lot of faults but none of them represented real problems;
- There were no consistent differences in the total number of AI vs. MF faults found by the subjects within each team. This was major improvement from the results in our previous study [5] where, the subjects using the NLtoSTD-BB method consistently found larger number of MF faults than the AI faults. This is a positive result that the improved heuristics were able to find both types of faults when constructing the STD-BBs.
- For each document, the average number of faults was calculated for each team by dividing the total number of unique faults by the number of subjects who inspected the document. The results showed that teams 1 through 5 found an average of 15, 12, 18, 9, and 22 faults respectively. This demonstrates an improvement in the performance of student teams from the first study [5] when using the original NLtoSTD-BB method (teams found an average of 7 faults at most) as well as a improvement when considering the inspection results in [5] when using the fault checklist method (an average of 5 faults).

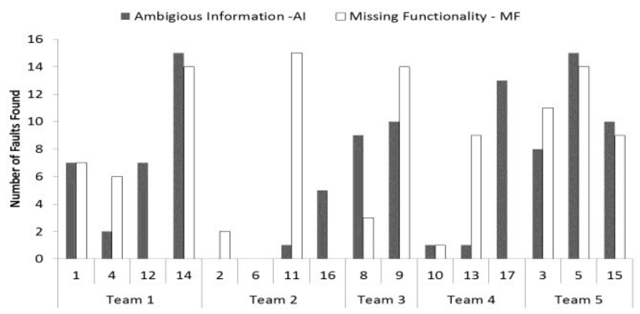


Fig. 4. Number of AI and MF Faults found by Subjects using NLtoSTD-BB

Therefore, based on these results, fault detection ***effectiveness*** of the NLtoSTD-BB method has improved from its first evaluation. Additionally, the distribution of faults is more consistent across both fault types (AI and MF). The revised NLtoSTD-BB heuristics were able to highlight hidden ambiguities in individual requirements which are otherwise not detected when performing a traditional inspection process.

Regarding the ***efficiency*** (faults/hour), the student teams found an average of 19, 7, 10, 10, and 26 faults per hour respectively. This is also an improvement in comparison to the results from the first study [5]. The high efficiency values reported in this study validate the ease of use of the revised NLtoSTD-BB method. Therefore, the NLtoSTD-BB is an effective and efficient method for verifying NL requirements.

B. *RQ 2: Fault Detection during the STD Creation*

The translation of BBs into STD can highlight the ambiguities and incompleteness in the NL requirements by examining the gaps (or disconnections) and inconsistent path traversals in the STD, and to identify the *inconsistencies* in the requirements that are not a focus during the NLtoSTD-BB.

To investigate the validity of this step, we counted the number of new MF, AI and INC faults reported by each subject after the creation and analysis of STD for their respective documents. The result on the number of new AI, MF, and Inc faults found by each subject belonging to a team is shown in Fig. 5. Interestingly (Fig. 5), each subject found at least one new fault (AI or MF or Inc) after creating the STD. As expected, a larger number of "*Inc*" faults are found during this step as compared to the AI and MF faults. This is also consistent across all the teams. The subjects felt it was easy to observe the inconsistencies when looking at a complete STD as opposed to translating individual NL requirements (one at a time) to STD-BBs. Since, the loading of BBs to create the STD is an automated process (using a tool); it is not surprising that subjects were able to find additional faults by focusing their attention on examining the STD and recording faults.

To better understand the *effectiveness* of STD-BBtoSTD, we compared the percentage contribution of the STD-BBtoSTD relative to the overall NLtoSTD for each team. This was done by dividing the # of unique faults found during STD-BBtoSTD by sum of total # of unique faults found during NLtoSTD-BB and STD-BBtoSTD combined. The student teams 1-5, after the creation of STD, found 18%, 22%, 14%, 23%, and 12% of total faults respectively. To further verify the usefulness of the creation and analysis of STD, a one-sample t-

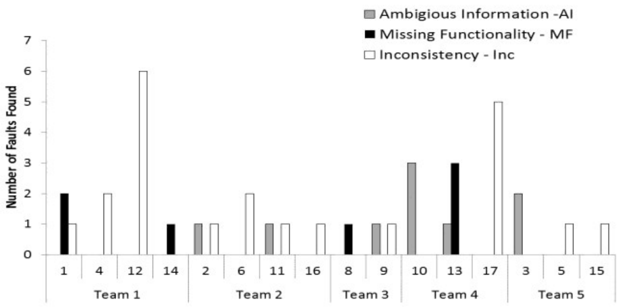


Fig. 5. Number of Faults found after the Creation of STD using the STD-BBs

test was run separately for each team to determine whether the number of faults found during re-inspection using STD was significantly greater than zero (0). The result was found to be statistically significant ($p < 0.001$) and indicate a benefit of creating STD using the BBs, for finding requirement faults.

C. Difficulties Faced by Subjects Using the NLtoSTD Method

The qualitative data collected during this study evaluated the usability of the NLtoSTD-BB method. To do that, the participants were asked to rate the difficulty level for finding the "*Entity*", "*Initial Status*", "*Changed Status*", "*How is Status Changed*", and "*Conditions for Change*" for NL requirements.

Using a 5-point likert scale (1-very difficult to 5-very easy), the participants rated the difficulty level for each of the five elements of an STD-BB. A One-sample Wilcoxon Signed-Rank test determined whether the medians ratings were significantly greater than 3 (midpoint of the scale). The results showed that the NLtoSTD-BB method received positive ratings (i.e., median value greater than or equal to three), but not statistically significant. The subjects also rated the difficulty level during the construction of STD and analyzing the constructed STD for faults. The results showed that the STD creation received significantly positive ratings ($p < 0.05$).

The complete NLtoSTD method was also evaluated using the feedback from subjects on the following seven attributes: *Simplicity*, *Ease of Understanding*, *Ease of Use*, *Intuitiveness*, *Comprehensiveness*, *Usefulness*, and *Ease of finding faults*. Each subject rated the attributes on a 5-point scale. The results from Wilcoxon Signed-Rank test revealed that the NLtoSTD received significantly positive ratings on *Ease of Understanding*, *Ease of Use*, and *Ease of finding faults*.

Overall, the subjects felt that the NLtoSTD process helped them understand the major problems in requirements, and that the effort spent during the NLtoSTD inspection process was worthwhile. The potential improvements regarding the tool and the guidance to help analyze the STD diagram will be implemented in future evaluations.

V. Discussion Of Results

RQ 1: The NLtoSTD-BB method helped inspectors find inherent ambiguities and incompleteness in requirements. The comparison of the results against the previous research results [5] revealed that the subjects were able to find larger number of total faults (on average), and the distribution of faults across fault types (MF and AI) was more consistent. The results also showed that the NLtoSTD-BB method helped find the faults faster (i.e., efficiency) when compared to the results in [5].

RQ 2: Based on the results, additional MF and AI fault types are uncovered during the examination of STD constructed from the BBs. In particular, the creation of STD aids inspectors at detecting a large number of "Inc" that are otherwise not apparent when looking at individual requirements. The construction of STD is useful for overall inspection effectiveness using the complete NLtoSTD method.

RQ 3: The subjects provided insights in to the use of the NLtoSTD method and improvements that can help improve the performance in future studies. The subjects mentioned that the tool should guide the NLtoSTD-BBs translation and should at least highlight parts of STD that are completely disconnected. We plan to make this process as much automated as possible without losing the promise of inspections.

VI. Conclusion And Future Work

Based on these results, the NLtoSTD method is effective method to detect AI, MF, and Inconsistency fault types during an inspection of NL requirements. We also identified the areas of improvement that would benefit the performance of subjects using the method. Our future work would include more replications, with a classic control group design so that we can understand how many faults found during the Phase III are solely due to the STD creation and not just due to the re-inspection. We also wish to automate as much of the heuristics as possible, including the NLtoSTD building block portion. The STD analysis could be automated as well, using a reasoning engine written in a logic language such as Prolog, and this has already been achieved to a certain degree.

References

[1] B. Boehm and V. Basili. Software fault reduction top 10 list. IEEE Computer, pages 135–137, January 2001.

[2] B. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

[3] B. Brykczynski, A survey of software inspection checklists, ACM SE Notes, 24(1):82,1999.

[4] C. Damas, B. Lambeau, P. Dupont, and A. Lamsweerde, Generating annotated behavior models from end-user scenarios," TSE, 31(12):1056-1073, 2005.

[5] D. Aceituna, H. Do, G. Walia, and S. Lee. Evaluating the use of model-based requirements verification method: A feasibility study. *EmpiRE*, 2011, pages 13-20, August 30, 2011.

[6] D. Popescu, S. Rugaber, N. Medvidovic, and D. Berry, Reducing ambiguities in requirements specifications via automatically created object-oriented models, Monterey Workshop on Computer Packaging, pp. 103-124, 2007.

[7] F. Chantree, B. Nuseibeh, A. de Roeck, and A. Willis. Identifying nocuous ambiguities in natural language requirements. RE, pp 59–68, 2006.

[8] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong, Orthogonal fault classification - A concept for in-process measurements. TSE, 18(11): 943-956, 1992.

[9] S. Sakthivel. Survey of requirements verification techniques. Journal of Information Technology, pp. 68-79, 1991.

[10] L. Kof, R. Gacitua, M. Rouncefield, and P. Sawyer, Ontology and model alignment as a means for requirements validation, ICSC, pp. 46-51, 2010.

[11] D. Barry. Ambiguity in natural language requirements documents. *Lecture Notes in Computer Science, LNCS*, volume 5320, pages 1-7, 2008.

[12] E. Letier, J. Kramer, J. Magee, and S. Uchitel, Monitoring and control in scenario-based requirements analysis. In *Proceedings of the 27th International Conference on Software Engineering*, pages 382–391, 2005.